DART

Dart 语言介绍

现代的 Web 编程语言

大纲

1. 当前的Web开发
2. Dart 语言介绍
3. Dart 语法特性

Web应用的优点

- 无需安装
- 增量式开发
- 自动升级
- 跨平台
- 设计模式好

当前的Web开发

- Web 开发越来越复杂:项目大，团队成员多
- 前端功能越来越多:CS→BS→CS
- 大型应用维护和协作难
- 性能弱，启动速度慢
- 工具支持弱:查找/替换

当前的Web开发

- JavaScript 中一些不好的部分
- 程序结构不明显
- 缺少一些基础功能的支持，如模块
- 碎片化，共享复用难，如模块的不同实现AMD和CommonJS
- 性能不可预测(高效的部分只是语言的一个子集)

JavaScript对class的模拟

```
// MooTools
var Cat = new Class({
  initialize: function(name){
    this.name = name;
  }
});
```

```
// Prototype
var Person = Class.create({
  initialize: function(name) {
    this.name = name;
  }
});
```

```
// Dojo
declare("mynamespace.MyClass", null, {
  // Custom properties and methods here
});
```

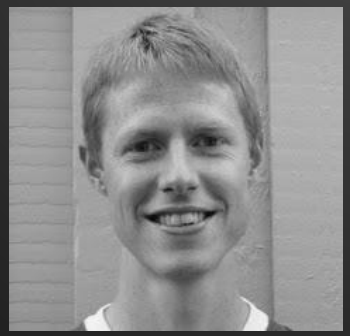
```
// ExtJS
Ext.define('Ext.Window', {
  extend: 'Ext.Panel',
  requires: 'Ext.Tool'
});
```

Web 中只有一种语言？

- Web 语言单一，OS 语言丰富
- 单一语言难以适合所有工作
- JavaScript 成了其它语言的目标代码
- 语言自身的进化？
  - “三岁看小，七岁看老”
  - 兼容性
  - 各语言的进化史

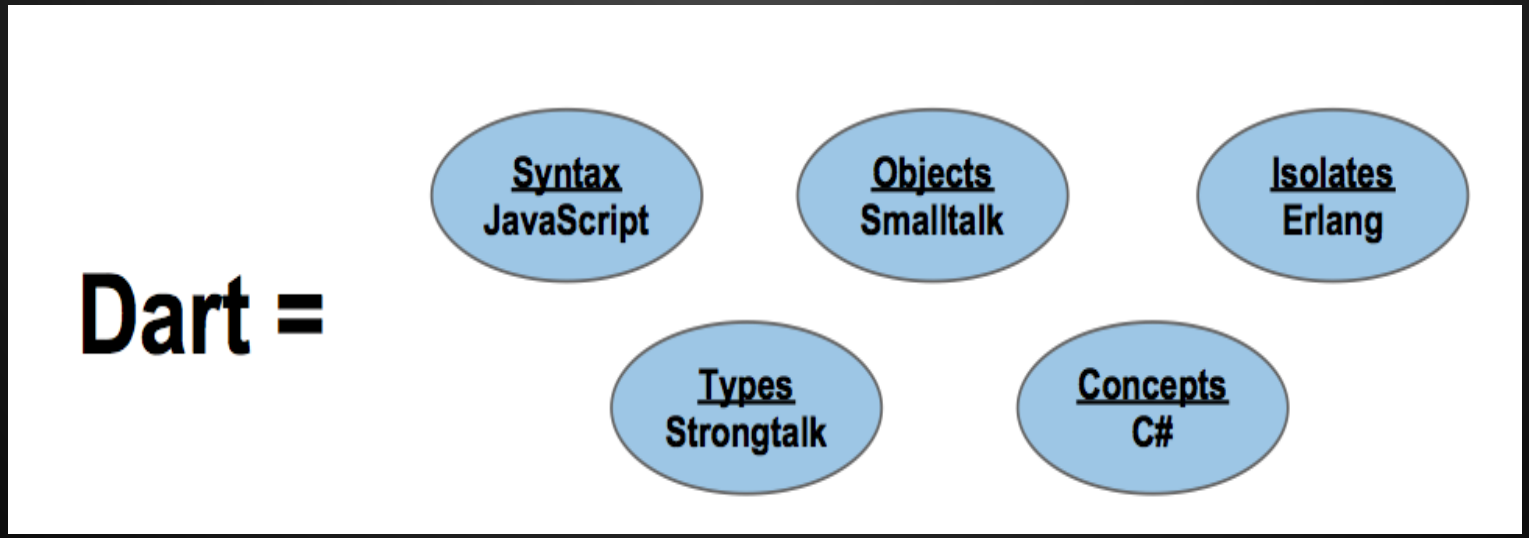
Dart 语言的诞生

- 2011年10月在GOTO大会上对外宣布
- 最初由 Chrome V8 团队打造
- 主要成员包括：Lars Bak、Kasper Lund、Gilad Bracha等（V8、Strongtalk、Java Hotspot、Self、JavaScript…）

Dart 是什么样语言

- 简单、熟悉的面向对象语言
- 基于类，单继承、多实现
- 熟悉的语法和恰当的词法作用域
- 可选的静态类型
- 单线程和基于Isolate的并发



Dart 语言的目标

- 结构化并且灵活的Web语言
- 适合从小型到大型的项目
- 熟悉、自然，易于学习
- 简单、有生产力
- 高性能、快速启动
- 适合各种设备的Web环境
- 运行于所有主流浏览器

Dart 代码示例

```
class Point {
  var x, y;
  Point(this.x, this.y);
  operator +(other) => new Point(x + other.x, y + other.
y);
  toString() => "($x,$y)";
}

main() {
  var p = new Point(2, 3);
  print(p + new Point(4, 5));
}
```

Dart 语言的组成

Dart 不仅是一种语言，包括：

- 语言规范
- Dart VM
- 丰富的类库
- dart2js：Dart → JavaScript 编译器
- Dartium：Chrome + Dart VM
- 包管理 Pub：pub.dartlang.org
- IED：Dart Editor 等
- 分析工具

Dart 两种运行模式

1）检查模式（checked）

检查类型匹配，及早发现问题，但性能差

`T x = o` 等价于 `assert(o == null || o is T)`

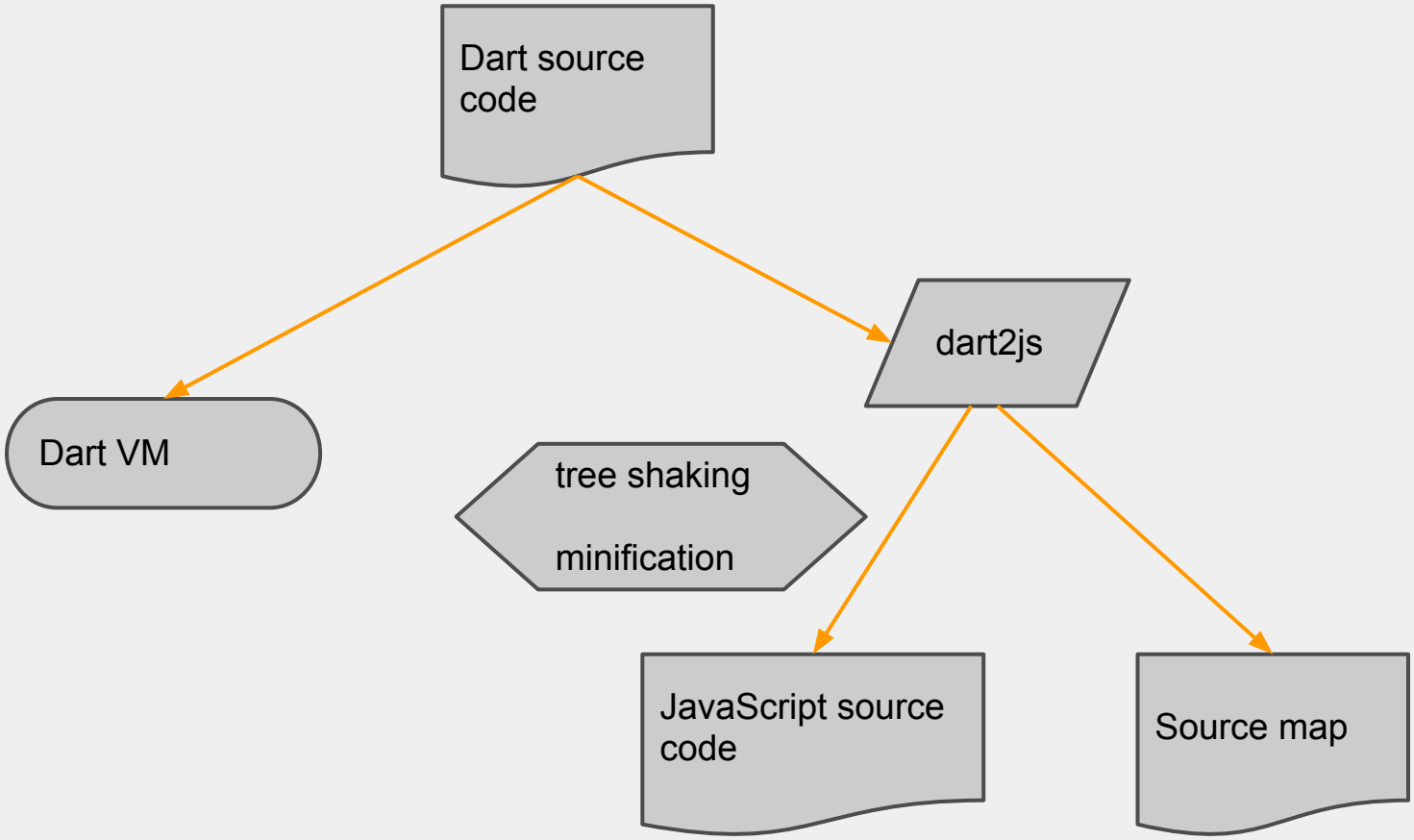
2）生产模式（production）

不检查类型，性能好

Dart 运行环境

- 客户端（Web+HTML5，PC+Mobile）
  - 支持 Dart 的浏览器：VM直接运行
  - 不支持 Dart 的浏览器：dart2js编译为JavaScript
- 服务器端环境
  - Web Server
  - File、Socket、Process

Dart 运行环境


Dart source code
Dart VM
dart2js
tree shaking
minification
JavaScript source code
Source map

在网页中嵌入Dart

Dart 的 MIME 类型:application/dart

辅助 js:dart.js

```
<script type="application/dart"
        src="app.dart"></script>

<script src="packages/browser/dart.js">
</script>
```

Dart 的浏览器支持

dart2js 支持：

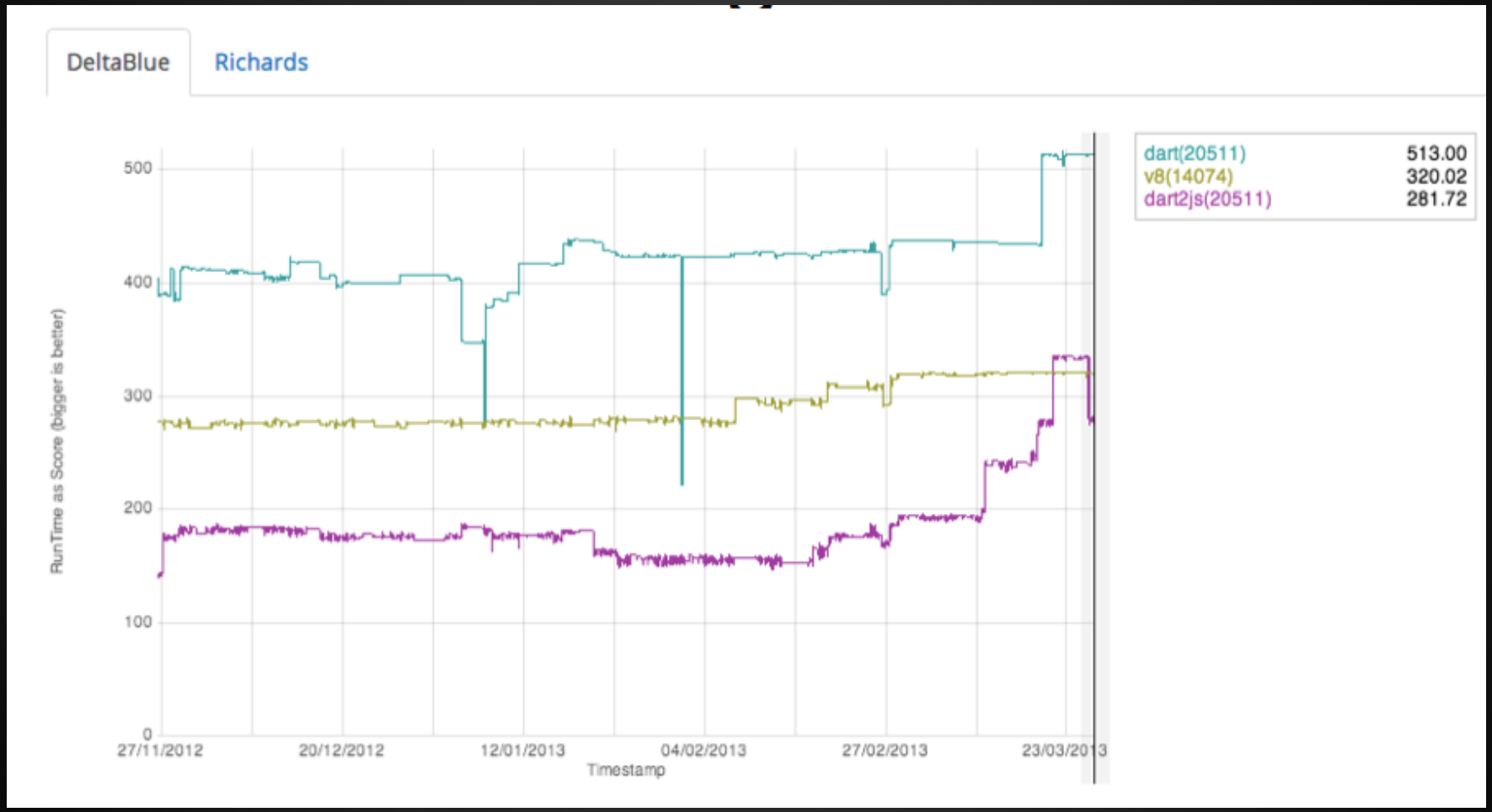
- Internet Explorer 9 and 10
- Safari
- Firefox
- Chrome
- iOS Safari
- Chrome for Android

原生的 Dartium：集成调试+快速迭代

高性能

- 语言的设计影响性能
- 使用snapshot启动速度快10倍
- 运行时不能改变对象结构有利于VM优化
- Dart VM 性能已超越 JavaScript V8
- ……

高性能



http://www.dartlang.org/performance/

独立VM的好处

Dart 有自己的VM

CoffeeScript、TypeScript 没有

独立VM的优点：

- 开发调试的连续性：没有编译过程，IDE紧密集成
- 独立 VM 带来性能上的根本改进
- dart2js 不能从根本上提高性能

Dart 语法特性

- 纯面向对象：类
- 范型
- 函数、闭包
- 库：import
- 可选类型
- 词法作用域
- 异步与并发编程
- getter 和 setter
- 方法级联调用
- 操作符重载
- Markdown注释
- 支持 Mixin
- 基于Mirror的反射
- 不支持 eval
- ……

Dart 故意让人感到熟悉

- 复用已有语言的成熟表示
- 没必要与众不同，并且易于学习
- 一致性
- 不追求代码的短小而带来麻烦
- 有坑的地方一定会有人跳（这和聪明无关）

关于类型

- 动态类型与静态类型
- 强类型与弱类型
- 各自的优势与工具支持
  - 类型应用：变量、参数、返回值
- Dart 结合动态类型与静态类型的优点

类型的好处

- 类型即文档，表明程序意图，提供概念框架
- 避免特定的变量命名模式或注释方式
- 适合范型
- 良好工具的支持
- 更早发现问题(类型检查)：
  - 检查模式运行
  - 静态代码分析
- 类型完全是可选的。

可选类型

- Dart 是动态语言，类型不影响运行时语义
- 可选类型不影响运行时语义：
  - 不能基于类型的方法重载
  - 基于类型的初始化: int i 不等价 int i = 0
- 运行时执行并不依赖与类型检查
- 可选类型可以理解为一种类型断言机制，而非静态类型系统。

可选类型示例:无类型

```
class Point {
  var x, y;
  Point(this.x, this.y);
  operator +(other) => new Point(x + other.x, y + other.
y);
  toString() => "($x,$y)";
}

main() {
  var p = new Point(2, 3);
  print(p + new Point(4, 5));
}
```

可选类型示例:有类型

```
class Point {
  num x, y;
  Point(this.x, this.y);
  Point operator +(Point other) =>
                    new Point(x + other.x, y + other.y);
  String toString() => "($x,$y)";
}

main() {
  Point p = new Point(2, 3);
  print(p + new Point(4, 5));
}
```

可选类型示例:

无类型标注时, 参数都代表什么？写文档/看源码/猜

```
recalculate(origin, offset, estimate) {
  ...
}
```

有类型标注时, 参数一目了然

```
num recalculate(Point origin, num offset,
                {bool estimate: false}) {
  ...
}
```

工具的支持

语言设计影响工具的支持

Dart Editor支持:

- 代码导航:提示大型项目的工作效率
- 代码自动完成
- 重构:查找/替换太痛苦
- 调试:与Dartium集成
- 静态代码分析:错误和警告
- 快速修正

Dart 减轻提前设计的负担

设计的重要原则：避免基础部分的不兼容修改，否则导致大量依赖代码的调整。

两难：预留足够的灵活性？可能导致过度设计。这加重了前期设计者的负担。

面向对象的选择

1）prototype vs class

2）运行时不能修改类结构

- 减少不必要麻烦
- 有利于性能优化

面向对象:类和接口的统一

- 类是类，也是接口。
- 每个类都有一个隐式接口
- 继承一个类
- 实现一个或多个类(接口)
- 抽象类和接口
- 接口可以有默认实现

面向对象:类和接口的统一

```
abstract class Shape {
  num perimeter(); //计算周长
}

class Rectangle implements Shape {
  final num height, width;
  Rectangle(num this.height, num this.width);
  num perimeter() => 2*height + 2*width;
}

class Square extends Rectangle {
  Square(num size) : super(size, size);
}
```

面向对象:接口的默认实现

```
// In Java
List list = new ArrayList();
// In Dart
List list = new List();
```

只需定义一个(工厂)构造函数，并返回其默认实现。

基本类型

Dart中所有事物都是对象

num: int , double

String，boolean，List，Map

支持 final、const

boolean 只有 true 才是 true

原则：语言不应该为了短小而引入更多麻烦

Dart 中只有 true 和 false：概念简单，防范错误

隐式转换的问题：陷阱与记忆负担

函数：First-class citizen

- 函数：作为变量、参数、返回值
- 定义函数无需 ~~function~~ 关键字
- return 返回值，否则返回null
- 快捷定义：`f() => true;`
- 函数的类型是Function，也可以 typedef 类型
- 可选参数：命名可选参数和位置可选参数

函数:函数、闭包和别名

```
typedef num adder(num); // alias

adder makeAdder(num n) {
  return (num i) => n + i;
}

main() {
  adder add2 = makeAdder(2);
  print(add2(3)); // 5
}
```

位置可选参数

按照参数位置管理的可选参数

定义:`fn(a, [b, c])`
默认值:`fn(a, [b=2, c=3])`
使用:

- `fn(1)`
- `fn(1, 2)`
- `fn(1, 2, 3)`

命名可选参数

按照参数名管理的可选参数，使用时与位置无关。使用时要额外指名，但代码可读性更好。

定义:`fn(a, {b, c})`
默认值:`fn(a, {b:2, c:3})`
使用:

- `fn(1)`
- `fn(1, c:3)`
- `fn(1, b:2, c:3)`

this不改变:所见即所得

```
class AwesomeButton {
  int awesomeDial;
  ButtonElement elem;
  AwesomeButton(this.elem) {
    elem.onClick.listen((e) => this.crankTheAwesome());
  }
  crankTheAwesome() {
    awesomeDial = 11;
  }
}
```

字段和方法的统一

- getter/setter 其实就是方法调用
- 只是以字段的形式使用
- 字段可以认为是自动实现的getter/setter
- 无需提前设计

getter/setter：字段和方法的统一

```
class Rectangle {
  num left, top, width, height;

  num get right              => left + width;
      set right(num value)   => left = value - width;
  num get bottom             => top + height;
      set bottom(num value)  => top = value - height;

  Rectangle(this.left, this.top, this.width, this.height);
}

var rect = new Rectangle(3, 4, 20, 15);
print(rect.left);
print(rect.bottom);
rect.top = 6;
rect.right = 12;
```

方法级联（Method Cascade）

- 在同一个对象上连续调用多个方法
- 传统实现：API设计者必须特意返回this
- Dart：语言级的支持，不必预先设计

方法级联调用示例

```
var button = query('#button');
button.text = 'Click to Confirm';
button.classes.add('important');
button.onClick.listen((e) => window.alert('Confirmed!'));
```

```
query('#button')
    ..text = 'Click to Confirm'
    ..classes.add('important')
    ..onClick.listen((e) => window.alert('Confirmed!'));

// or Build a new Button
var b = new ButtonElement()
  ..id = 'awesome'
  ..classes.add('important')
  ..onClick.listen(rockOn);
```

构造函数:自动初始化字段

```
class Point {
  num x, y;
  Point(num x, num y) {
    this.x = x;
    this.y = y;
  }
}
```

消除重复

```
class Point {
  num x, y;
  Point(this.x, this.y);
}
```

构造函数:命名构造函数

```
class Point {
  num x, y;
  Point(this.x, this.y);
  Point.polar(num r, num t) {
    x = r * cos(t);
    y = r * sin(t);
  }
}

new Point(2, 3); new Point.polar(3, 0.21);
```

构造函数:工厂构造函数

```
class Symbol {
  final String name;
  static Map<String, Symbol> _cache = {};
  factory Symbol(String name) {
    if (_cache.containsKey(name)) {
      return _cache[name];
    } else {
      final symbol = new Symbol._internal(name);
      _cache[name] = symbol;
      return symbol;
    }
  }
  Symbol._internal(this.name);
}
new Symbol('something');
```

基于Future的异步API

- Dart是单线程模式，没有线程间共享状态
- UI和程序在同一个线程中执行
- 运行时负责维护一个事件循环（event loop），所有工作以队列的方式依次执行
- Future or Promise，而不是 callback
- Future 代表稍后才会获得的对象
  - 回调函数冗长
  - 太多嵌套回调的代码难读

同步读文件

```
print('at the beginning');
var contents = file.readAsStringSync();
print(contents);
print('at the end');
```

输出:

```
at the beginning
文件内容
at the end
```

异步读文件

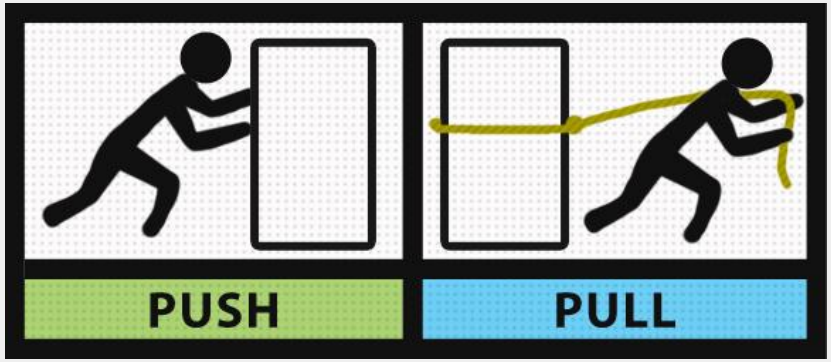
```
print('at the beginning');
file.readAsString()
    .then((String content){
      print(content);
    });
//or file.readAsString().then(print);
print('at the end');
```

输出:

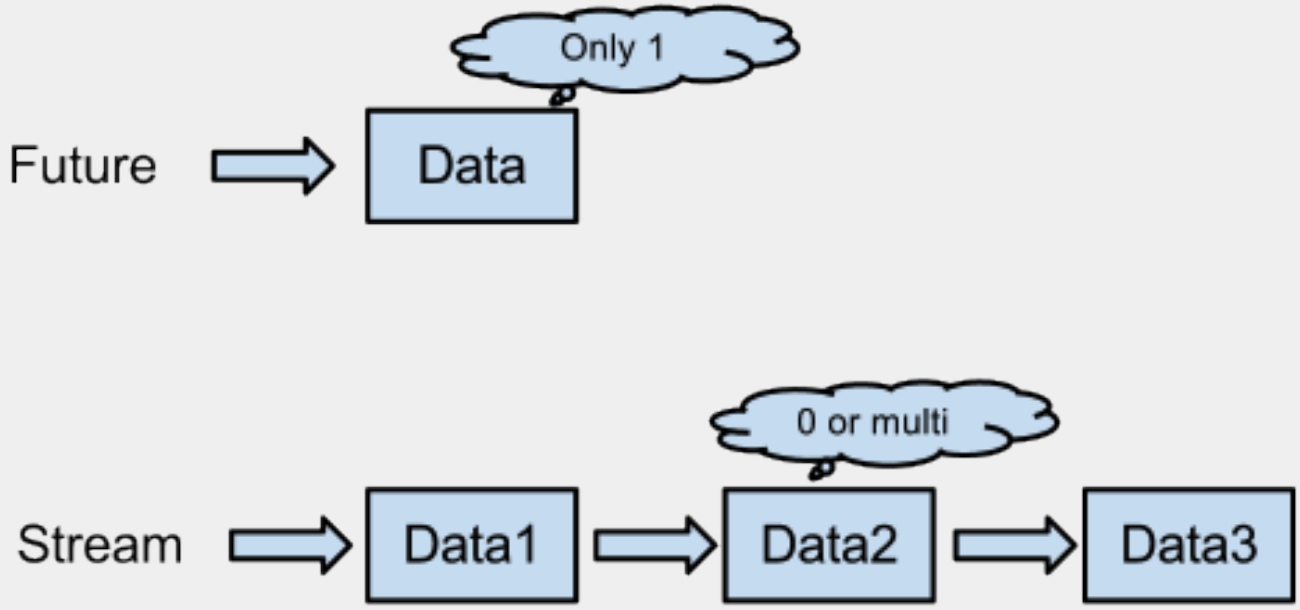
```
at the beginning
at the end
文件内容
```

Stream:统一的异步事件处理模型

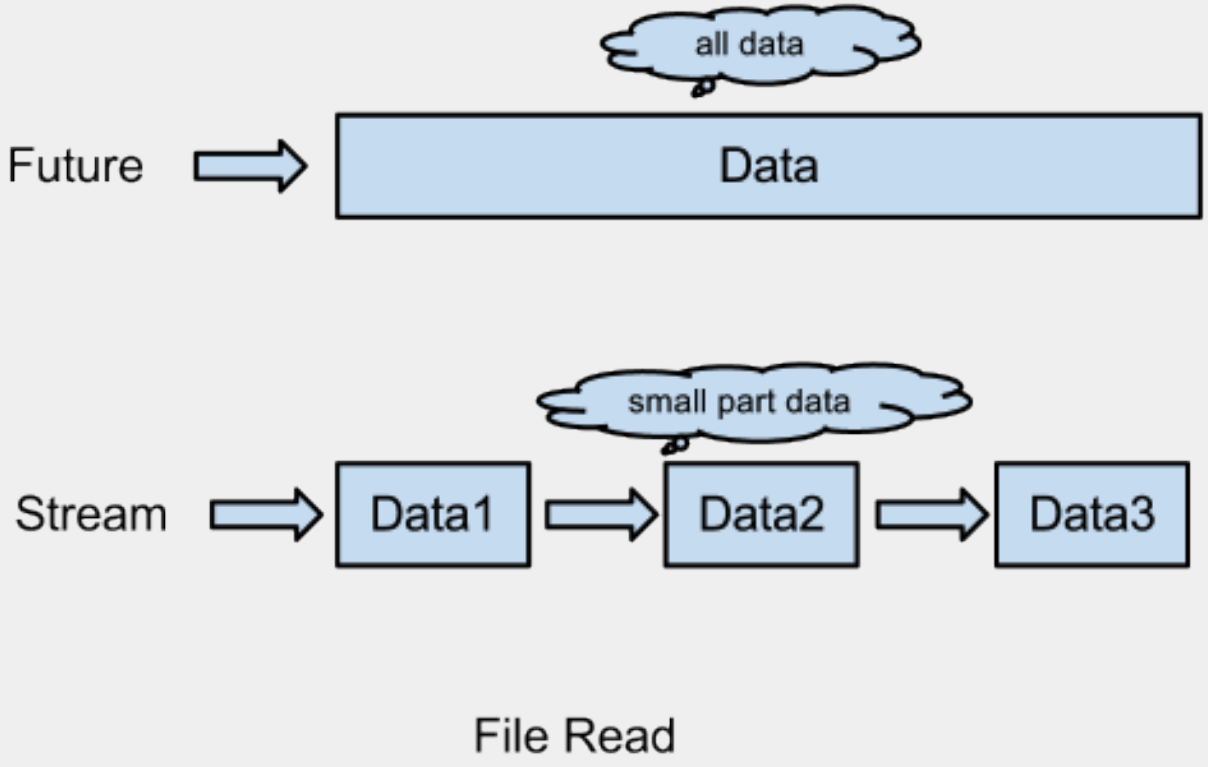
Stream ：异步版的集合（Iterable）



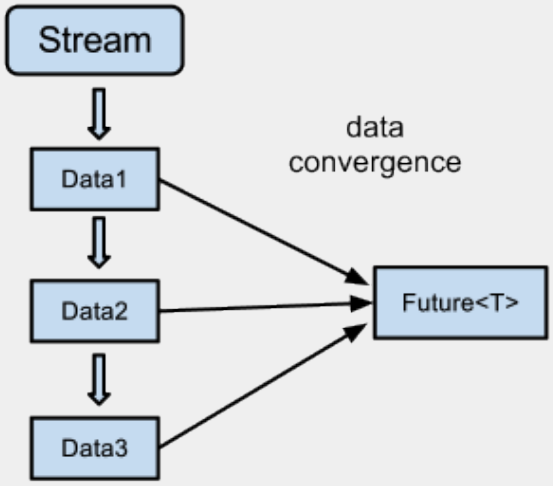
Stream 与 Future

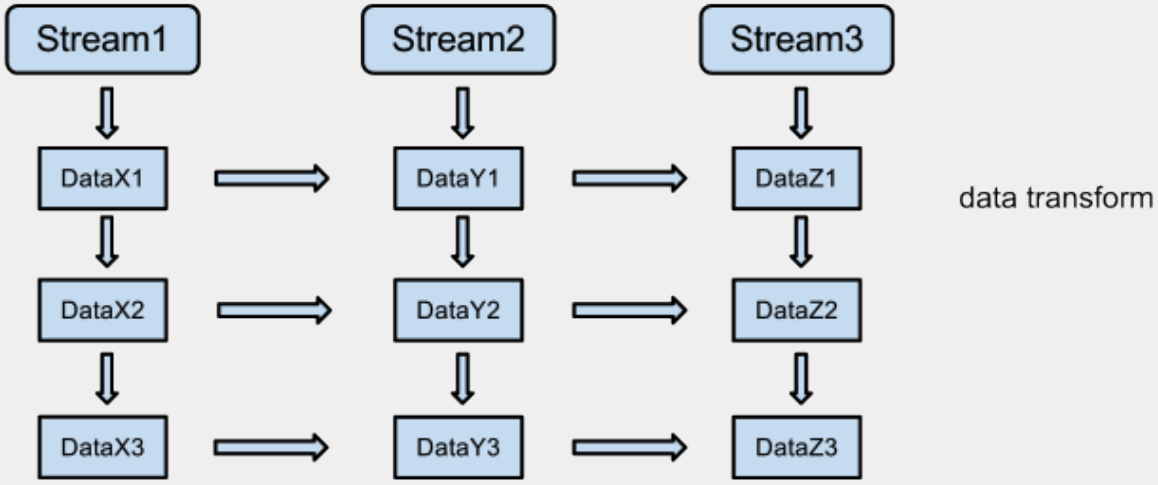


Stream 与 Future



数据流的两种处理模式


Stream
data
convergence
Data1
Data2
Data3
Future<T>


Stream1
Stream2
Stream3
DataX1
DataX2
DataX3
DataY1
DataY2
DataY3
DataZ1
DataZ2
DataZ3
data transform

并发编程:Isolate

- 受Erlang Actor模式的启发的一种轻量级的执行单元
- Isolate之间完全隔离，没有任何共享状态，无需同步
- 所有Dart代码都在Isolate中运行，包括main
- Isolate之间仅通过端口上的消息传递通信
- 消息在发送前被复制

并发编程:Isolate

```
import 'dart:isolate';
echo(){
  print('start echo');
  port.receive((msg, replyTo){
    if(replyTo != null){
      replyTo.send('hello $msg');
    }
  });
}
main() {
  var sendPort = spawnFunction(echo);
  sendPort.call('hanguokai').then((reply) {
    print(reply);
  });
}
```

并发编程:Isolate

```
void costlyQuery() {
    port.receive((sql, reply) {
        //block code
        executeSql(sql);
        reply.send();
    });
}

main() {
    var sendPort = spawnFunction(costlyQuery);
    sendPort.call("select * from dual").then(callback);
    print("Called before executeSql finishes");
}
```

Mixin

Mixin 是介于子类和超类之间的层次

方法优先级从右向左：self > M1 > M2 > S

```
class X extends S with M2, M1
```

Mixin 类的限制

- 不能声明构造函数
- 超类是Object
- 不能调用 super

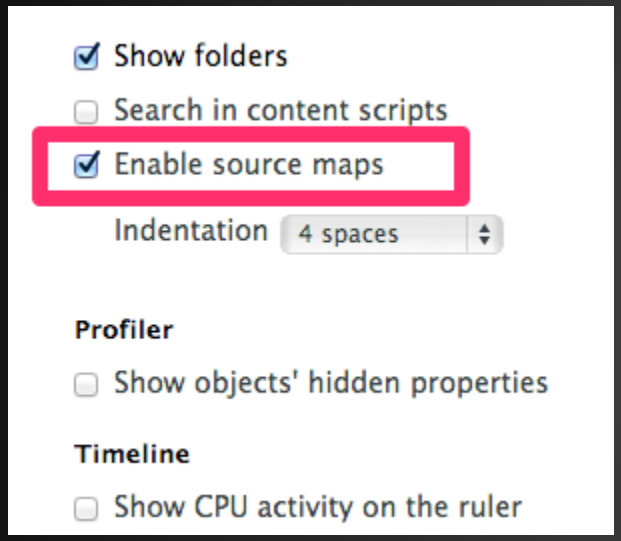
Mixin

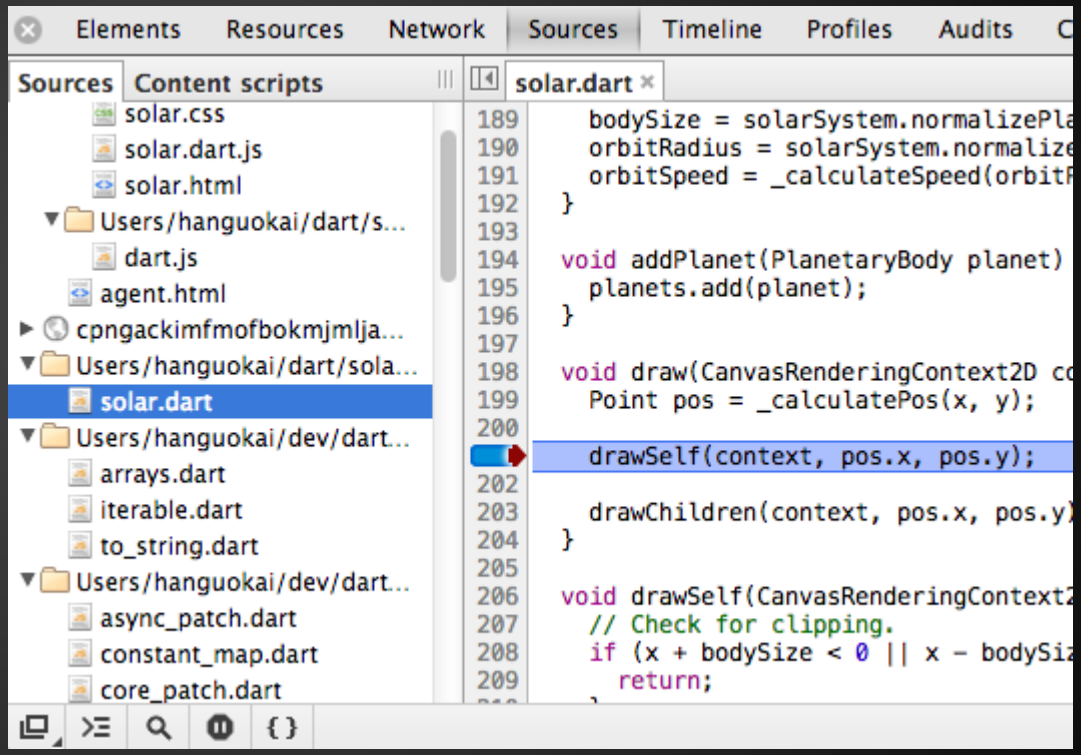
```
abstract class Persistence {
  void save() {
    ...
  }
}

class Person extends Object with Persistence
{

}
```

Source Map 调试





库的组织

import 语句

导入内置库:`import 'dart:html';`

导入Pub包:`import 'package:unittest/unittest.dart';`

library 语句

```
library math;
```

part 和 partof 语句

export 语句

作用域和私有概念

Dart 小结

- 简洁、熟悉的语法
- 坑少，工具支持好
- 性能好
- 项目可伸缩
- Web 和 Server 编程
- 库正在逐渐丰富

Dart 资源

官方网站：http://dartlang.org

邮件列表、Google Plus、Stackoverflow

开源项目：https://code.google.com/p/dart/

子项目：https://github.com/dart-lang

Pub：http://pub.dartlang.org

国内交流：

- 微博：@Dart语言
- ChinaGDG 论坛：http://chinagdg.com
- 下载镜像：http://dart.hanguokai.com/

谢谢！Q & A

@hanguokai
@Dart语言
hanguokai.com

Dart 现在就能用！